MISC

November 16, 2018
by Niklaus Schiess

Dumping Decrypted Documents from a North Korean PDF Reader

This is a writeup about how to use Frida to dump documents from a process after they have been loaded and decrypted. It's a generic and very effective approach demonstrated on a piece of software from North Korea.

Some time ago we received an ISO file which was a dump of a CD-ROM from North Korea. The only information we got was that it included a document viewer and various PDF documents. I started to dump the content of the ISO in order to analyze what the reader was actually doing by mounting it:

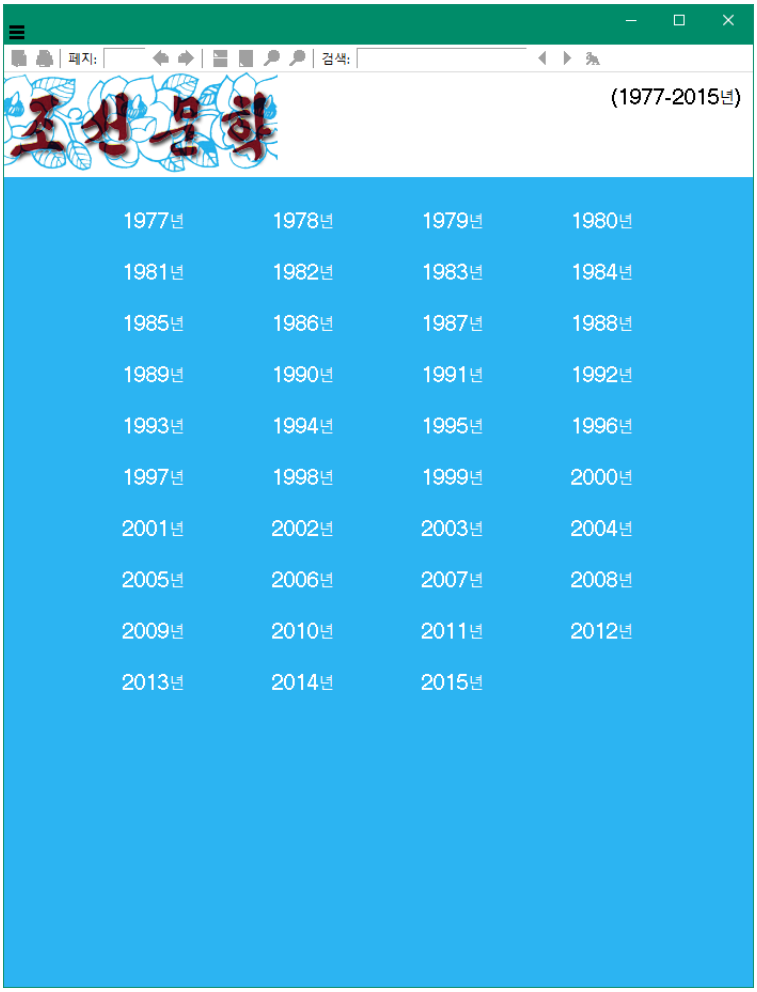
```
$ sudo mount -o loop docs.iso /mnt
```

The ISO included a single file called *KLMviewer.exe*. I tried to run it on various versions of Windows including XP, 7 and 10 but the program just did nothing. I started to analyze it in IDA and quickly found the reason why the program did nothing. The following screenshot shows a snippet of the *WinMain()* function.

```
143  if ( GetDriveTypeW(0) != 5 )
144    goto LABEL_108;
145  GetVolumeInformationW(0, &VolumeNameBuffer, 0x80u, 0, 0, 0, 0, 0);
146  GetFileAttributesExW(L"data", 0, &FileInformation);
147  FileTimeToSystemTime(&FileTime, &SystemTime);
148  wsprintfW(
149    &v78,
150    L"%X",
151    (unsigned int)&unk_5A5A5A ^ 13 * (SystemTime.wDay + 100 * (SystemTime.wMonth + 100 * (SystemTime.wYear % 100))));
152  if ( wcsncmp(&VolumeNameBuffer, L"KLM-", 4u) )
153    goto LABEL_108;
```

WinMain() checks of *KLMviewer.exe*

The first line actually checks if the program has been started from a CD-ROM. The last check on the screenshot also checks if the CD-ROM's volume name matches "KLM-*". So I just inserted the ISO into a Windows VM and started the reader directly from the CD-ROM which indeed worked. It turned out that the program is actually a PDF reader that includes a lot of North Korean documents.



KLMviewer.exe main window

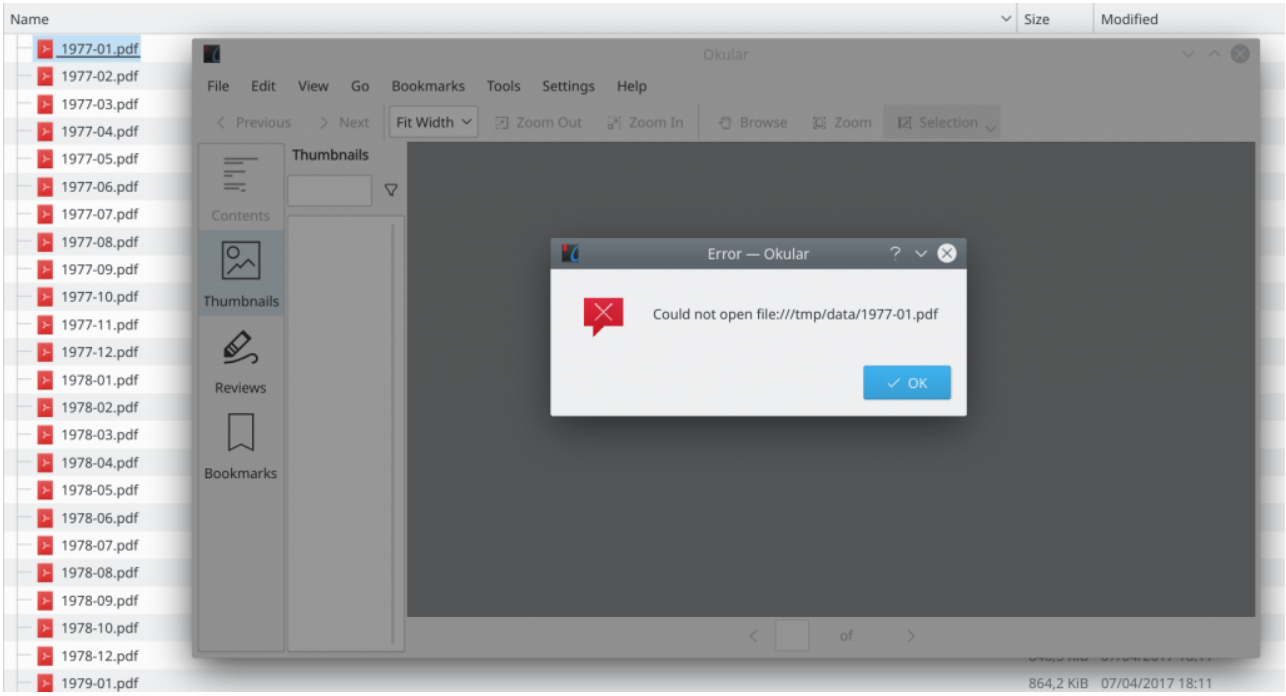
The viewer only allows to open one document at a time. It also has some weird properties when it comes to printing and copy&pasting: it only allows to print a single page and copy up to 1000 characters.

KLMviewer.exe with open document

The interesting part was that the reader seemed to include all of those PDF files, actually several hundred documents. But the reader was just ~6 MB large. One interesting piece of information from the initial screenshot of the *WinMain()* checks was that it referenced a "data" directory which was not on the ISO when I mounted it. It turns out that the directory was just hidden and my mount refused to make them visible. In order to make the directory visible you have to mount the ISO with the "unhide" option:

```
$ sudo mount -o loop,unhide docs.iso /mnt
```

After this the data directory became visible and it included all the PDF files. But the files were not readable.



Raw PDF files are not readable

Checking the header of the PDF files also showed that these are not valid PDF documents:

```
$ xxd 1977-01.pdf | head -n2
00000000: 80f5 e1e3 8894 8b92 a8af 8004 1660 72a8  .............`r.
00000010: af94 8595 85ca c7cf a8af 9999 8ae6 cacb  ................
```

A valid PDF document would start with the magic bytes "%PDF", which was not the case for these documents. So it seems like the PDFs are most likely encrypted and the PDF viewer loads and decrypts them as you open them.

Analyzing the PDF Reader

By analyzing the binary it quickly showed that the PDF reader is actually a fork of SumatraPDF, which is open source. The authors of *KLMviewer.exe* just added various checks and code paths that decrypt the PDFs on the ISO. They did not use the password/encryption feature of SumatraPDF.

In order to avoid reversing their encryption scheme we decided to use another approach: dumping the PDF files from the running process right after they have been loaded and decrypted. This process requires to find a function that gets a pointer to the memory location of the decrypted PDF file. At this point we started to read the code of SumatraPDF to find the point where the PDF files are loaded into memory but right before the engine parses the actual PDF file.

An interesting function that we found was *PdfEngineImpl::Load()* which calls *fz_open_file2()*:

```
fz_stream* fz_open_file2(fz_context* ctx, const WCHAR* filePath) {
    fz_stream* file = nullptr;
    int64_t fileSize = file::GetSize(filePath);
    // load small files entirely into memory so that they can be
    // overwritten even by programs that don't open files with FILE_SHARE_READ
    if (fileSize > 0 && fileSize < MAX_MEMORY_FILE_SIZE) {
        fz_buffer* data = nullptr;
```

```
            fz_var(data);
            fz_try(ctx) {
                data = fz_new_buffer(ctx, (int)fileSize);
                data->len = (int)fileSize;
                if (file::ReadN(filePath, (char*)data->data, data->len))
                    file = fz_open_buffer(ctx, data);
            }
            fz_catch(ctx) { file = nullptr; }
            fz_drop_buffer(ctx, data);
            if (file)
                return file;
        }

        fz_try(ctx) { file = fz_open_file_w(ctx, filePath); }
        fz_catch(ctx) { file = nullptr; }
        return file;
    }
```

fz_open_file2() gets the filename as a first argument and reads it into a *fz_buffer* object.

```
fz_buffer* data = nullptr;
```

The *fz_buffer* struct looks like this:

```
struct fz_buffer_s
{
        int refs;
        unsigned char *data;
        int cap, len;
        int unused_bits;
};
```

So in our data object the actual pointer to the document will be in data + 4 (after the integer refs, which is 4 bytes long) and the length of the buffer will be at data + 12. This data object will then be given to the *fz_open_buffer()* function, which is the perfect candidate to dump the decrypted PDF file, because the second argument is the data object which contains a pointer to the raw PDF file in memory.

After spending some time search the functions in the binary, we found the offsets of *fz_open_file2()* at *sub_167A90()* and *fz_open_buffer()* at *sub_1A50A0()*. Although the binary did not include symbols this was rather easy: We just checked the strings of the error messages to pinpoint the functions in the binary. After crosschecking the function arguments and what the function roughly does (e.g. which other functions are called) this was pretty efficient.

One thing that should be noted is that the binary uses the Microsoft fastcall calling convention, which passes the first two arguments via the ECX and EDX registers. This becomes important once you start hooking functions with Frida, because it doesn't recognize this by default.

Hooking the functions with Frida

We started to implement a simple Frida script that hooks both the *fz_open_file2()* and *fz_open_buffer()* functions. Hooking both is important because *fz_open_buffer()* gets the pointer to the PDF, but only *fz_open_file2()* knows the filename. This is required to save the documents with their proper filename. Also the first argument for both functions is a context object. We just used this to make sure that we are reading the same file in both functions.

We ended up with the following Frida script:

```
var baseAddr = Module.findBaseAddress('KLMviewer.exe');
console.log('KLMviewer.exe baseAddr: ' + baseAddr);
var fz_open_file2 = baseAddr.add(0x0073690);
var fz_open_buffer = baseAddr.add(0x00B50A0);

Interceptor.attach(fz_open_file2, {
    onEnter: function (args) {
        console.log('[+] Called fz_open_file2: ' + fz_open_file2);
        var ctx = ptr(this.context.ecx); // read first argument from ECX
        console.log('[+] Ctx: ' + ctx);
        var lpFileName = Memory.readByteArray(this.context.edx, 38); // read sec
        send({type: 'ctx-lpfilename', 'context': ctx}, lpFileName);
    }
});
Interceptor.attach(fz_open_buffer, {
    onEnter: function (args) {
        console.log('');
        console.log('[+] Called fz_open_buffer: ' + fz_open_buffer);
        var ctx = ptr(this.context.ecx); // read first argument from ECX
        console.log('[+] Ctx: ' + ctx);
        console.log('[+] data pointer: ' + this.context.edx); // read second arg
        var data_pointer = ptr(this.context.edx);
        var pdf = Memory.readPointer(data_pointer.add(4)); // data pointer is at
        console.log('[+] pointer to pdf: ' + pdf);
        var size = Memory.readU32(data_pointer.add(12)); // data length is at da
        console.log('[+] size of pdf: ' + size);
        var buf = Memory.readByteArray(pdf, size); // read the PDF document fron
        send({type: 'data', 'context': ctx}, buf);
    },
});
```

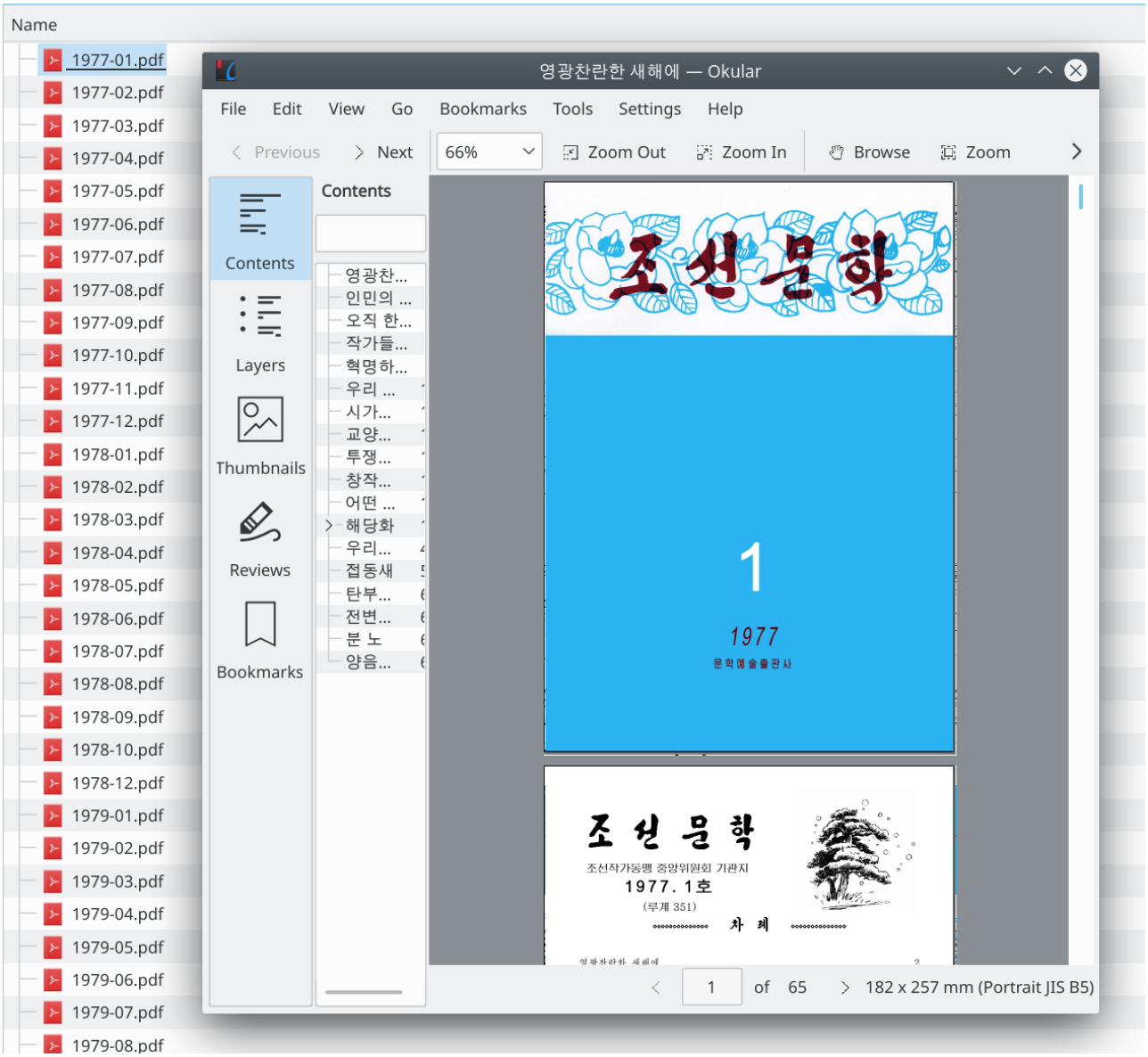
We then implemented a Python function for the incoming messages that checks the current context and writes the PDF files from memory to the file system if the context matches.

The full script is available here (requires Python 3 and Frida: *pip3 install frida*). In order to extract the documents one just has to start the PDF reader and get it's PID (e.g. with the Windows task manager or

Process Explorer) and give it to the script:

```
$ python3 KLMviewer_frida.py 1234
```

As soon as a document will be opened in the PDF reader it will be dumped to a file on the file system in the current directory. The following screenshot shows how we are able to access the PDFs after dumping them with Frida:



Reading the extracted PDF documents

This writeup is just an example how easy it can be to dump memory from processes after e.g. it has been loaded and decrypted. We didn't have to analyze the actual crypto implementation to get the decrypted PDF files and we did not have to alter the executable at all.

Thanks to Birk for the Frida support.

Cheers!

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

POST COMMENT